Data Mining

National Crime Victimization Survey

Varun Nadgir Vijay Kesari

Data Introduction

Before loading any data into R, our first step was to read the provided NCVS codebook and become familiar with the context of the study. This allowed us to build an understanding of the types of responses and what we would need to be mindful of in the dataset. We found that the responses are all categorical-for example, the field V2022 is for "Location of Phone" and has possible values ranging from 1 to 9. According to the codebook, the responses will fall into one of the following categories:

- Valid values are usually in the range of smaller integer numbers.
- Explicit "don't know" responses are ones that are clearly offered as a response type
- Blind "don't know" responses are not offered but still accepted for most questions
- Refused response means the respondent chose not to answer
- "Residue" responses usually mean something is missing and are usually identified by '8', '98', '99', '998', etc. based on the length of an accepted response
- "Out of scope/universe" responses are those that are not applicable to the question

There are several reasons why a response might be blank, causing some ambiguity between blind "don't know" responses and refused responses. However, we can handle residue and out of scope responses in our preprocessing.

Our initial dataset contains 4947 tuples of 204 variables before any preprocessing. In addition to the 203 training variables, the target variable is *o_bullied* - it has a value of 1 if the student experienced bullying and 0 if they did not. Because we have a positive and negative class, we will be building binary classifiers.

Data Mining Tools

- modeest
- dplyr
- e1071
- caret
- rsample
- rpart
- randomForest
- xgboost
- glmnet
- pROC
- class
- nnet

Preprocessing

Since there are results that indicate non-answers, we will be avoiding including those in our training. There are also several columns that are heavily populated by a single response. Our first step is to target low variance and noisy data. One way to address these in bulk is to simply filter by number of factors. Variables with only a few levels will not contain Residual or Out of universe responses

Classification Algorithms

- Algorithm 1: Logistic Regression
- Algorithm 2: Decision Tree/Random Forest
- Algorithm 3: Gradient Boost (XGBoost)

- Algorithm 4: Naive Bayes
- Algorithm 5: K-Nearest Neighbors
- Algorithm 6: Neural Network
- Algorithm 7: SVM

Algorithm 1: Logistic Regression

Logistic Regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes). It estimates the probability that a given instance belongs to a certain class.

Algorithm 2: Decision Tree/Random Forest

A Decision Tree is a flowchart-like tree structure where an internal node represents a feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome. The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in a recursive manner called recursive partitioning. A Random Forest is a meta-estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Algorithm 3: Gradient Boosting (XGBoost)

XGBoost (Extreme Gradient Boosting) is an efficient and scalable implementation of gradient boosting. It provides a parallel tree boosting that solves many data science problems in a fast and accurate way. XGBoost is widely used for supervised learning problems, where we use the training data with multiple features to predict a target variable.

Algorithm 4: Naive Bayes

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naïve) independence assumptions between the features. They are among the simplest Bayesian network models but coupled with kernel density estimation, they can achieve higher accuracy levels.

Algorithm 5: K-Nearest Neighbors (KNN)

The K-Nearest Neighbors algorithm (KNN) is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. It classifies a data point based on how its neighbors are classified. KNN stores all available cases and classifies new cases by a majority vote of its k neighbors.

Algorithm 6: Neural Network

Neural Networks consist of layers of interconnected nodes or neurons, which are inspired by biological neural networks. Each connection can transmit a signal from one neuron to another. The receiving neuron processes the signal and then signals downstream neurons connected to it. Neural networks can model complex non-linear relationships.

Algorithm 7: Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, each data item is plotted as a point in n-dimensional space (where n is the number of features you have) with the value of each feature being the value of a particular coordinate. Then, classification is performed by finding the hyper-plane that best differentiates the two classes.

Data Mining Procedure

```r
library(modeest)
library(dplyr)
library(e1071)
library(caret)
library(rsample)
library(rpart)
library(randomForest)
library(xgboost)
library(glmnet)
library(smotefamily)
library(pROC)
library(class)
library(nnet)

# 1) import initial data
data <- read.csv("project_dataset.csv")
```

First, to reduce the number of Residue and Out of universe responses, we realized we could not universally remove 8's and 9's from our data. Some fields have beyond 9 options, meaning that the values 8 and 9 are valid responses. Thinking about it logically, we concluded that if a field does not contain 7, then the responses of 8 or 9 *cannot* be valid, as there are not enough options. By turning these to NA, we have filtered out many of the invalid responses, though not all. Then, by setting a threshold for the number of acceptable NA values a field can have, we can further reduce the number of columns.

```r
# 2) list of columns containing the value 7
contains_7 <- apply(data==7, 2, any)

# if a column does not contain 7, then values of 8 and 9 MUST be invalid
data[, !contains_7] <- lapply(data[, !contains_7], function(x) ifelse(x >= 8, NA, x))

# 4) replace values greater than or equal to 98 with NA (out of universe)
data[data>=98] <- NA

# 5) set a threshold for number of NA values in a column
na_thresh <- 3000

# drop columns that have NA values over our threshold
columns_to_remove <- colSums(is.na(data)) > na_thresh

# this reduces us down to 152 training variables
data <- data[, !columns_to_remove]

# 6) apply mfv from modeest package on every column
data <- data %>%
  mutate_all(~ ifelse(is.na(.), mfv(., na_rm=TRUE), .))

# 7) identify columns near zero variance (heavily one level)
near_zero_var <- nearZeroVar(data[1:length(data)-1], freqCut=90/10)

# this reduces us down to 85 training variables
data <- data[ ,-near_zero_var]
```

After, we change all the columns to be factor, so that they are treated as having levels. By filtering for a

given number of levels, such as 2 or 3, we are able to test and debug our models at low runtime cost and then gradually scale up. It was not used in the final encoding (see comment block), but it helped in finding errors as we proceeded. We then create the dummy variables and encode the dataset, which expands the number of columns we have but allows us to use numerical analysis on our values.

We also experimented with over- and under-sampling of the classes in the training set using SMOTE from *smotefamily* and downSample and upSample from *caret*, since we have many more class 0 cases. Ultimately we decided to not use any method on our global data (see comment block), since the performance of the models was not being significantly improved and we felt that keeping the original set would help keep results authentic.

```
# 8) convert all variables to factors
data <- as.data.frame(lapply(data, as.factor))

# 9) filter factor variables with given number of levels (for test/debugging)
# data_filtered <- data[, sapply(data, function(col) length(levels(col)) %in% 2:9)]

# 10) create dummy vars and encode filter
# data_dummy <- dummyVars("~ .", data = data_filtered, fullRank = TRUE)
# data_encoded <- as.data.frame(predict(data_dummy, newdata = data_filtered))
# names(data_encoded)[names(data_encoded) == "o_bullied.1"] <- "o_bullied"

# 10) create dummy vars and encode original
data_dummy <- dummyVars("~ .", data = data, fullRank = TRUE)
data_encoded <- as.data.frame(predict(data_dummy, newdata = data))
names(data_encoded)[names(data_encoded) == "o_bullied.1"] <- "o_bullied"

# set seed for reproducibility
set.seed(123)

# 11) split 70-30
split <- initial_split(data_encoded, prop=0.75)
tr_encoded <- training(split)

# 12) apply oversampling on training for class 1
# tr_encoded <- SMOTE(tr_encoded[, -length(tr_encoded)],
                    # tr_encoded[, length(tr_encoded)])$data
# names(tr_encoded)[names(tr_encoded) == "class"] <- "o_bullied"
# tr_encoded$o_bullied <- as.numeric(tr_encoded$o_bullied)

ts_encoded <- testing(split)

# 13) remove columns of 0-only, whether in tr or ts
all_zero_col <- c(unname(which(sapply(tr_encoded, function(col) all(col==0)))),
                  unname(which(sapply(ts_encoded, function(col) all(col==0)))))
all_zero_col <- unique(all_zero_col)

if (length(all_zero_col)>0) {
  tr_encoded <- tr_encoded[, -all_zero_col]
  ts_encoded <- ts_encoded[, -all_zero_col]
}

write.csv(data_encoded, "preprocessed_data.csv")
write.csv(tr_encoded, "initial_train.csv")
```

```r
write.csv(ts_encoded, "initial_test.csv")

# show proportion of target variable in all datasets
table(data_encoded$o_bullied)/length(data_encoded$o_bullied)
```

```
##
##         0         1
## 0.7715787 0.2284213
```

```r
table(tr_encoded$o_bullied)/length(tr_encoded$o_bullied)
```

```
##
##         0         1
## 0.7722372 0.2277628
```

```r
table(ts_encoded$o_bullied)/length(ts_encoded$o_bullied)
```

```
##
##         0         1
## 0.7696039 0.2303961
```

Now that we have our main dataset and the train and test set, we can check the proportion of class 0 to class 1. As we can see, it is skewed towards the negative class, or N, which may pose issues when it comes to training and predicting, since our model will have seen fewer class 1 in training. If necessary, we can attempt to over- or under-sample our data to deal with the imbalance.

Model Results & Evaluation

By creating a function to calculate the performance measures based on a given confusion matrix, we can keep our calculations clean. By using class weights as function arguments, we can adjust as needed for each model we train, giving us flexibility. Additionally, we have a function for identifying factor levels that are not present in both tr and ts, since this may happen for some splits. See **Appendix 1.1** for the full *calculate_measures* function.

We have also included ROC plot in the Appendix for any model where we had to determine a decision threshold, as it helped us to find a suitable value. By plotting our targets of Class 0 TPR of 75% and Class 1 TPR of 65%, we were able to see if those were attainable through tuning.

Model 1: Logistic Regression

```r
# 5-fold cv
ctrl <- trainControl(method="cv", number=5)

# train model
m <- train(o_bullied ~ ., data = tr_encoded, method = "glm",
           family = "binomial", trControl = ctrl)

# stop scientific notation
options(scipen=999)
```

```r
# compute class probabilities
m.pred <- predict(m, ts_encoded[, -length(ts_encoded)])

# change 0,1 to N,Y and set as factor
pred <- factor(ifelse(m.pred>=0.3, "Y", "N"))
target <- factor(ifelse(ts_encoded$o_bullied==1, "Y", "N"))

# create confusion matrix with Y as positive class
cm <- confusionMatrix(data=pred, reference=target, positive="Y")

# use default weights 1:1
measures <- calculate_measures(cm$table)

roc_model1.1 <- roc(as.numeric(target), as.numeric(pred))
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.8004	0.4456	0.8571	0.8004	0.8278	0.6774	0.3323	0.7437
Class Y	0.5544	0.1996	0.454	0.5544	0.4992	0.6774	0.3323	0.7437
Avg.	0.6774	0.6613	0.6556	0.6774	0.6635	0.6774	0.3323	0.7437

This model appears to slightly favor predicting the negative class, which we expected might be an issue. To help with this, we can introduce regularization to our logistic regression model to reduce some overfitting. Additionally, we can consider lowering the threshold for classification to see if having more accurate positive TPR has a severe, negative tradeoff.

```r
# set target back to numeric
tr_encoded$o_bullied <- as.numeric(as.character(tr_encoded$o_bullied))

# create training/response matrix
X <- model.matrix(o_bullied ~ ., data = tr_encoded)[, -1]
y <- tr_encoded$o_bullied

# cv grid
cv <- cv.glmnet(X, y, alpha = 1)

# train model with optimal lambda
m <- glmnet(X, y, alpha = 1, lambda = cv$lambda.min)


# Make predictions on the test set
ts_X <- model.matrix(o_bullied ~ ., data = ts_encoded)[, -1]
m.pred <- predict(m, newx = ts_X, s = cv$lambda.min, type = "response")

# change 0,1 to N,Y and set as factor
pred <- factor(ifelse(m.pred >= 0.27, "Y", "N"))
target <- factor(ifelse(ts_encoded$o_bullied==1, "Y", "N"))

# create confusion matrix with Y as positive class
cm <- confusionMatrix(data=pred, reference=target, positive="Y")

# use default weights 1:1
```

```
measures <- calculate_measures(cm$table)

roc_model1.2 <- roc(as.numeric(target), as.numeric(pred))
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.7532	0.3404	0.8808	0.7532	0.812	0.7064	0.3664	0.7316
Class Y	0.6596	0.2468	0.4444	0.6596	0.5311	0.7064	0.3664	0.7316
Avg.	0.7064	0.6468	0.6626	0.7064	0.6715	0.7064	0.3664	0.7316

It seems that lowering the threshold and adding regularization has helped quite a bit, as TPR and FPR for both classes has reached our targets.

Model 2: Decision Tree / Random Forest

```
# train the decision tree model
m <- rpart(o_bullied ~ ., data = tr_encoded)

# predict probabilities for the positive class
m.pred <- predict(m, ts_encoded[, -length(ts_encoded)])

# change 0,1 to N,Y and set as factor
pred <- factor(ifelse(m.pred >= 0.25, "Y", "N"))
target <- factor(ifelse(ts_encoded$o_bullied == 1, "Y", "N"))

# create confusion matrix with Y as the positive class
cm <- confusionMatrix(data = pred, reference = target, positive = "Y")

# use default weights 1:1
measures <- calculate_measures(cm$table)

roc_model2.1 <- roc(as.numeric(target), as.numeric(pred))
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.7195	0.4596	0.8395	0.7195	0.7749	0.6299	0.231	0.6783
Class Y	0.5404	0.2805	0.3658	0.5404	0.4363	0.6299	0.231	0.6783
Avg.	0.6299	0.685	0.6026	0.6299	0.6056	0.6299	0.231	0.6783

This model behaves slightly worse than the first - its predictions also favor the N class. We can instead use a random forest and adjust the threshold once again.

```
# train random forest
m <- randomForest(o_bullied ~ ., data=tr_encoded, ntree=100)

m.pred <- predict(m, ts_encoded[ , -length(ts_encoded)])

# change 0,1 to N,Y and set as factor
pred <- factor(ifelse(m.pred>=0.3, "Y", "N"))
```

```r
target <- factor(ifelse(ts_encoded$o_bullied==1, "Y", "N"))

# create confusion matrix with Y as positive class
cm <- confusionMatrix(data=pred, reference=target, positive="Y")

# use default weights 1:1
measures <- calculate_measures(cm$table)

roc_model2.2 <- roc(as.numeric(target), as.numeric(pred))
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.7489	0.3825	0.8674	0.7489	0.8038	0.6832	0.3268	0.7187
Class Y	0.6175	0.2511	0.4241	0.6175	0.5029	0.6832	0.3268	0.7187
Avg.	0.6832	0.6584	0.6457	0.6832	0.6533	0.6832	0.3268	0.7187

By giving up TPR in the negative class, we get quite a bit of improvement in the positive TPR. However, this is still not enough to reach our targets.

Model 3: Gradient Boost / XGBoost

```r
# create xgb matrix
xgb_matrix <- xgb.DMatrix(data=as.matrix(tr_encoded[ , -length(tr_encoded)]),
                          label=tr_encoded$o_bullied)

# train binary model
m <- xgboost(data=xgb_matrix, nrounds=100, objective="binary:logistic", verbose=0)

# predict and evaluate
xgb_predictions <- predict(m, as.matrix(ts_encoded[ , -length(ts_encoded)]))
xgb_predictions <- ifelse(xgb_predictions >= 0.2, "Y", "N")
xgb_predictions <- as.factor(xgb_predictions)
xgb_cm <- confusionMatrix(data=xgb_predictions, reference=target, positive="Y")
measures <- calculate_measures(xgb_cm$table)

roc_model3.1 <- roc(as.numeric(target), as.numeric(pred))
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.7626	0.4035	0.8633	0.7626	0.8098	0.6795	0.3241	0.7243
Class Y	0.5965	0.2374	0.4293	0.5965	0.4993	0.6795	0.3241	0.7243
Avg.	0.6795	0.6602	0.6463	0.6795	0.6545	0.6795	0.3241	0.7243

This model performs decently, but the threshold started to trend too low and we were hesitant to set it any lower.

Model 4: Naive Bayes

```
m <- cv.glmnet(as.matrix(tr_encoded[, -length(tr_encoded)]), tr_encoded$o_bullied,
               family = "binomial", alpha = 1, nfolds = 15)

# Predict class probabilities
m.pred <- predict(m, newx = as.matrix(ts_encoded[, -length(ts_encoded)]),
                  s = "lambda.min", type = "response")

# Convert probabilities to class labels
pred <- factor(ifelse(m.pred >= 0.242, "Y", "N"))
target <- factor(ifelse(ts_encoded$o_bullied == 1, "Y", "N"))

# Create confusion matrix with Y as the positive class
cm <- confusionMatrix(data = pred, reference = target, positive = "Y")

# Calculate evaluation measures
measures <- calculate_measures(cm$table)

vroc_model4.1 <- roc(as.numeric(target), as.numeric(pred))
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.7521	0.3439	0.8796	0.7521	0.8109	0.7041	0.3624	0.73
Class Y	0.6561	0.2479	0.4421	0.6561	0.5282	0.7041	0.3624	0.73
Avg.	0.7041	0.6479	0.6608	0.7041	0.6696	0.7041	0.3624	0.73

This is another good model that performs as well as our second logistic regression model. Tried number of combinations for nfolds = 13, 15, 17, 20, 23, 30. The TPRs for Class_N and Class_Y are just right above the targets for nfolds=15.

Model 5: KNN

```
tr_knn <- downSample(x = tr_encoded[, -length(tr_encoded)],
                                y = factor(tr_encoded$o_bullied),
                                yname = "o_bullied")

m.pred <- knn(train = tr_knn[, -length(tr_knn)],
              test = ts_encoded[, -length(ts_encoded)],
              cl = tr_knn[, length(tr_knn)],
              k = 5)

# Convert class labels to N/Y
pred <- factor(ifelse(m.pred==1, "Y", "N"))
target <- factor(ifelse(ts_encoded$o_bullied == 1, "Y", "N"))

# Create confusion matrix with Y as the positive class
cm <- confusionMatrix(data = pred, reference = target, positive = "Y")

measures <- calculate_measures(cm$table)
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.8141	0.6035	0.8184	0.8141	0.8162	0.6053	0.2093	0.7179
Class Y	0.3965	0.1859	0.3897	0.3965	0.393	0.6053	0.2093	0.7179
Avg.	0.6053	0.6974	0.604	0.6053	0.6046	0.6053	0.2093	0.7179

Likely due to the closeness of the data, KNN was struggling to identify the Y class. To address this, we introduced a downSample step from the *caret* library to help identify class Y. We also tried implementing it with different k Neighbours = 3, 7, 6, 5. It improved the performance by a lot but not enough to reach our targets.

Model 6: Neural Network

```
# Neural network parameters
size <- 2  # Number of units in the hidden layer
decay <- 0.1  # Weight decay (regularization term)

train_data <- tr_encoded[, -length(tr_encoded)]
test_data <- ts_encoded[, -length(ts_encoded)]
train_labels <- as.numeric(tr_encoded[, length(tr_encoded)])
test_labels <- as.numeric(ts_encoded[, length(ts_encoded)])

train_data <- scale(train_data)
test_data <- scale(test_data)

# Fit the neural network model
nn_model <- nnet(train_data, train_labels, size = size, decay = decay, maxit = 200, trace=FALSE)

# Predict on the test set
nn_predictions <- predict(nn_model, test_data)

# Convert class labels to N/Y
pred <- factor(ifelse(nn_predictions>=0.22, "Y", "N"))
target <- factor(ifelse(ts_encoded$o_bullied == 1, "Y", "N"))

# Create confusion matrix with Y as the positive class
cm <- confusionMatrix(data = pred, reference = target, positive = "Y")

measures <- calculate_measures(cm$table)

roc_model6.1 <- roc(as.numeric(target), as.numeric(pred))
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.8078	0.5158	0.8395	0.8078	0.8233	0.646	0.2805	0.7332
Class Y	0.4842	0.1922	0.4299	0.4842	0.4554	0.646	0.2805	0.7332
Avg.	0.646	0.677	0.6347	0.646	0.6394	0.646	0.2805	0.7332

Again, due to closeness of data, the neural network struggles to recognize the Y class. We could not increase the number of units in the hidden layer as the weight limit is crossed if it is greater than 2.

Model 7: Support Vector Machine (SVM)

```
train_data_scaled <- tr_encoded[, -length(tr_encoded)] # excluding the target variable
test_data_scaled <- ts_encoded[, -length(ts_encoded)]  # excluding the target variable

train_labels <- factor(tr_encoded[, length(tr_encoded)])
test_labels <- factor(ts_encoded[, length(ts_encoded)])

# Train an SVM model on the training data
svm_model <- svm(train_data_scaled, train_labels, kernel = "radial", cost = 1, scale = TRUE)

# Predict using the SVM model on the test data
svm_predictions <- predict(svm_model, test_data_scaled)

# Convert probabilities to class labels based on the threshold
# Convert probabilities to class labels
pred <- factor(ifelse(svm_predictions == 1, "Y", "N"))
target <- factor(ifelse(ts_encoded$o_bullied == 1, "Y", "N"))

# Create confusion matrix with Y as the positive class
cm <- confusionMatrix(data = pred, reference = target, positive = "Y")

# Calculate evaluation measures
measures <- calculate_measures(cm$table)
```

	TPR	FPR	Precision	Recall	F-measure	ROC	MCC	Kappa
Class N	0.9779	0.8561	0.7923	0.9779	0.8754	0.5609	0.2351	0.7858
Class Y	0.1439	0.0221	0.6613	0.1439	0.2363	0.5609	0.2351	0.7858
Avg.	0.5609	0.7195	0.7268	0.5609	0.5559	0.5609	0.2351	0.7858

This model was completely imbalanced and utterly failed to recognise Class Y. Balancing the data might it better.

Conclusion

Out of our several models, the best performing would be Model 1.2. Using regularization with logistic regression fit the problem space well and plotting ROC was useful for tuning the model. After several iterations of data preprocessing, we found that too much and too little both negatively impacted performance significantly. Finding a balance of meaningful data while keeping the models performant and interpretable was the greatest challenge.

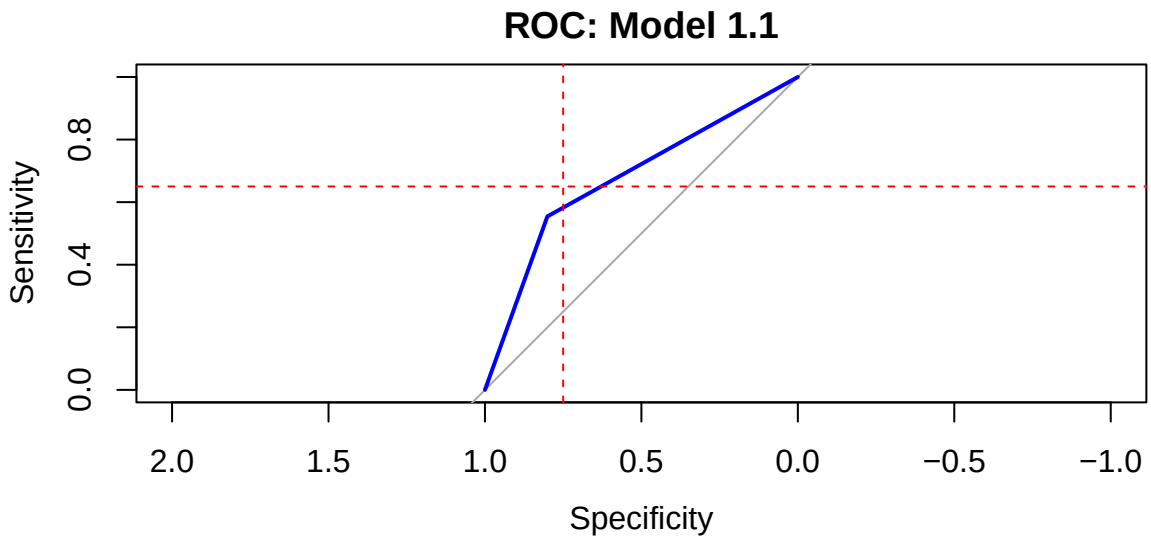
Appendix

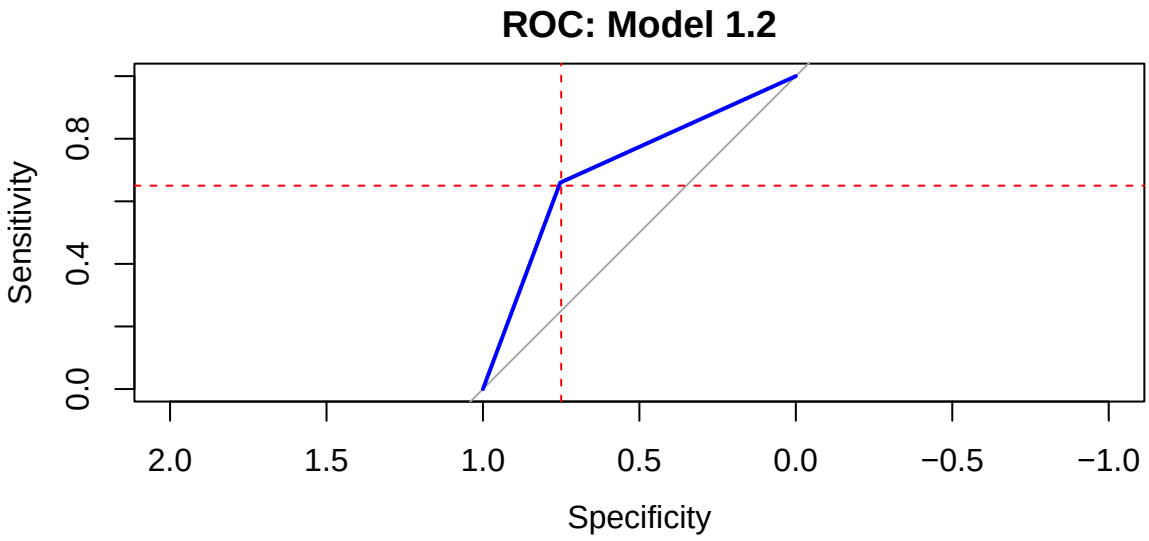
1.1:

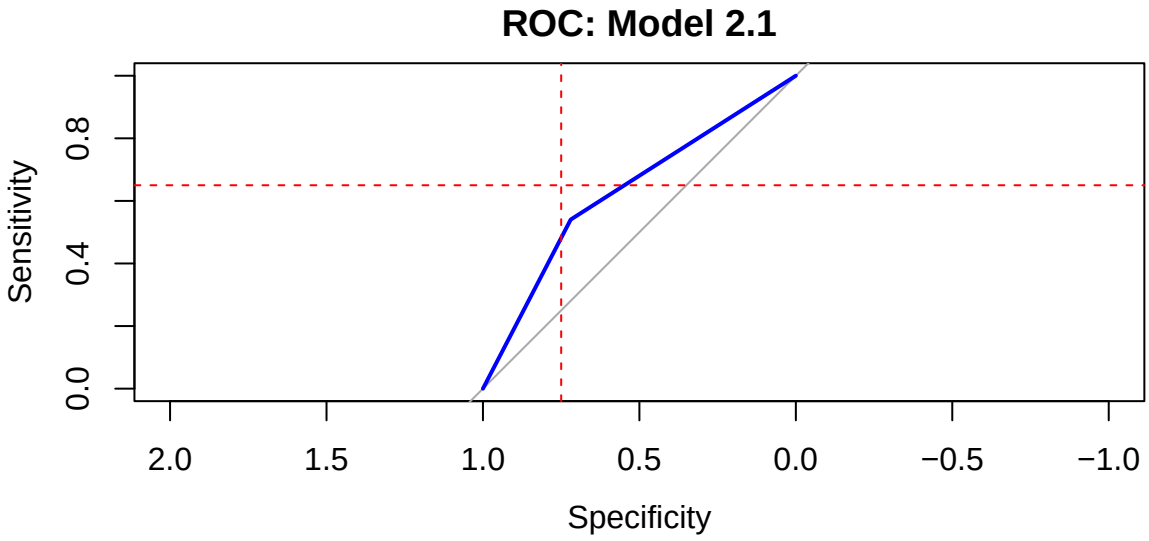
```
# define function to calculate performance measures based on given confusion matrix
calculate_measures <- function(cm, N_weight=1, Y_weight=1) {
  # target Positive Class
  tp_pos <- cm[2, 2]
  fp_pos <- cm[2, 1]
  tn_pos <- cm[1, 1]
  fn_pos <- cm[1, 2]

  # target Negative Class
  tp_neg <- cm[1, 1]
  fp_neg <- cm[1, 2]
  tn_neg <- cm[2, 2]
  fn_neg <- cm[2, 1]

  # calculate all Positive measures
  tpr_pos <- tp_pos/(tp_pos+fn_pos)
  fpr_pos <- fp_pos/(fp_pos+tn_pos)
  precision_pos <- tp_pos/(tp_pos+fp_pos)
  recall_pos <- tpr_pos
  f_measure_pos <- (2*precision_pos*recall_pos)/(precision_pos+recall_pos)
  roc_auc_pos <- (1+tpr_pos-fpr_pos)/2
  mcc_pos <- (tp_pos*tn_pos-fp_pos*fn_pos)/
    (sqrt(tp_pos+fp_pos)*sqrt(tp_pos+fn_pos)*sqrt(tn_pos+fp_pos)*sqrt(tn_pos+fn_pos))
  k_pos <- (tp_pos+tn_pos)/(tp_pos+tn_pos+fp_pos+fn_pos)

  # calculate all Negative measures
  tpr_neg <- tp_neg/(tp_neg+fn_neg)
  fpr_neg <- fp_neg/(fp_neg+tn_neg)
  precision_neg <- tp_neg/(tp_neg+fp_neg)
  recall_neg <- tpr_neg
  f_measure_neg <- (2*precision_neg*recall_neg)/(precision_neg+recall_neg)
  roc_auc_neg <- (1+tpr_neg-fpr_neg)/2
  mcc_neg <- (tp_neg*tn_neg-fp_neg*fn_neg)/
    (sqrt(tp_neg+fp_neg)*sqrt(tp_neg+fn_neg)*sqrt(tn_neg+fp_neg)*sqrt(tn_neg+fn_neg))
  k_neg <- (tp_neg+tn_neg)/(tp_neg+tn_neg+fp_neg+fn_neg)

  # calculate weighted averages
  weights <- c(N_weight, Y_weight)

  tpr_avg <- weighted.mean(c(tpr_neg, tpr_pos), weights)
  fpr_avg <- weighted.mean(c(fpr_neg, fpr_pos, weights))
  precision_avg <- weighted.mean(c(precision_neg, precision_pos), weights)
  recall_avg <- weighted.mean(c(recall_neg, recall_pos), weights)
  f_measure_avg <- weighted.mean(c(f_measure_neg, f_measure_pos), weights)
  roc_auc_avg <- weighted.mean(c(roc_auc_neg, roc_auc_pos), weights)
  mcc_avg <- weighted.mean(c(mcc_neg, mcc_pos), weights)
  k_avg <- weighted.mean(c(k_neg, k_pos), weights)

  # column names
```

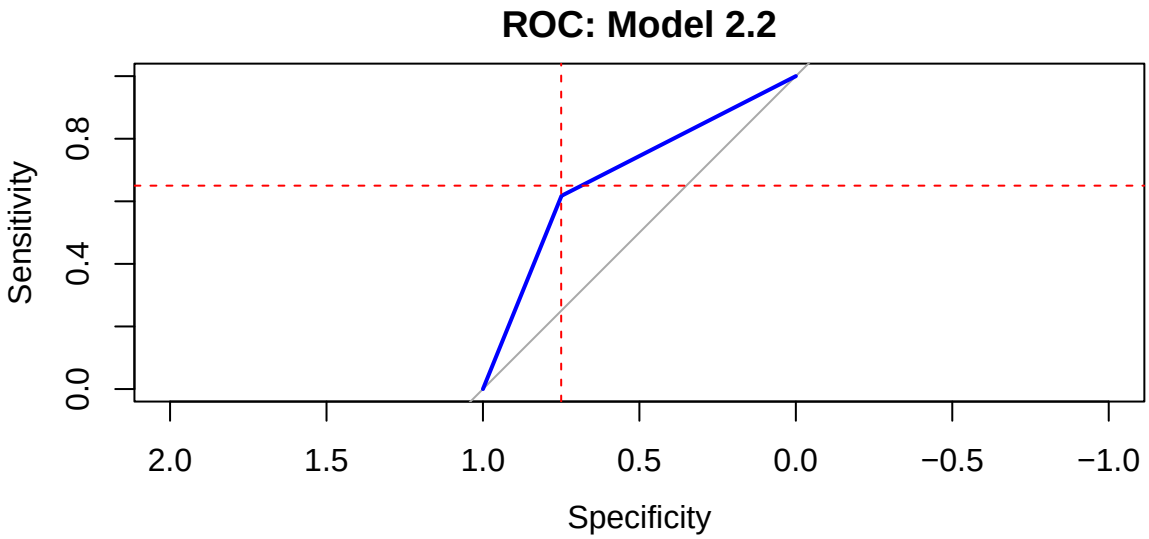
```
  measures <- c("TPR", "FPR", "Precision", "Recall", "F-measure", "ROC", "MCC", "Kappa")

  # row data
  positive_measures <- c(tpr_pos, fpr_pos, precision_pos, recall_pos,
                         f_measure_pos, roc_auc_pos, mcc_pos, k_pos)
  negative_measures <- c(tpr_neg, fpr_neg, precision_neg, recall_neg,
                         f_measure_neg, roc_auc_neg, mcc_neg, k_neg)
  weighted_average <- c(tpr_avg, fpr_avg, precision_avg, recall_avg,
                        f_measure_avg, roc_auc_avg, mcc_avg, k_avg)

  # create dataframe
  result_df <- data.frame(Class_N=negative_measures,
                          Class_Y=positive_measures,
                          Average=weighted_average)
  rownames(result_df) <- measures
  result_df <- round(result_df, 4)

  return(result_df)
}
```

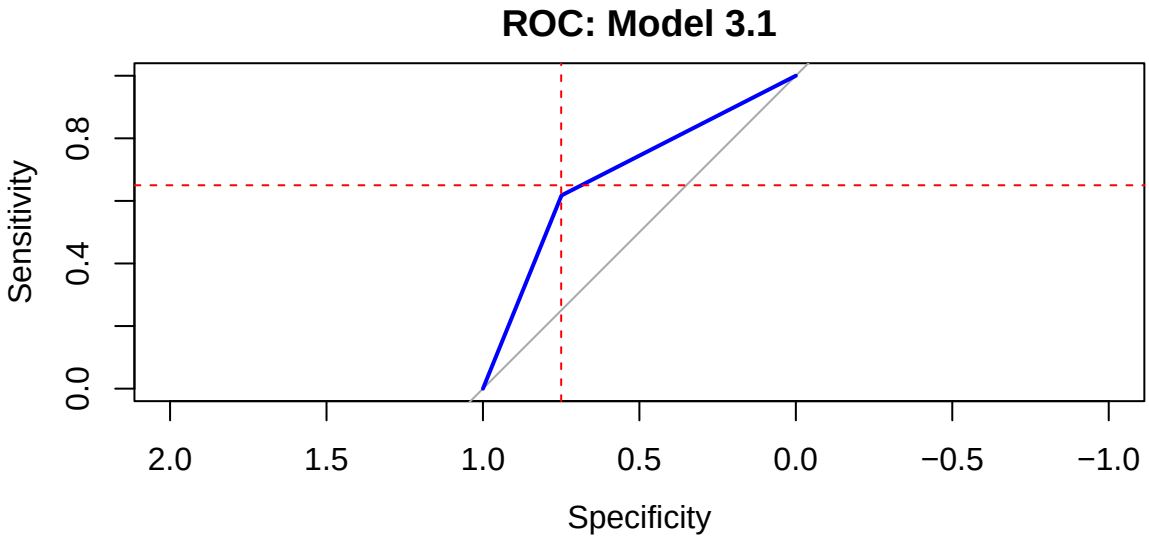
ROC For Models

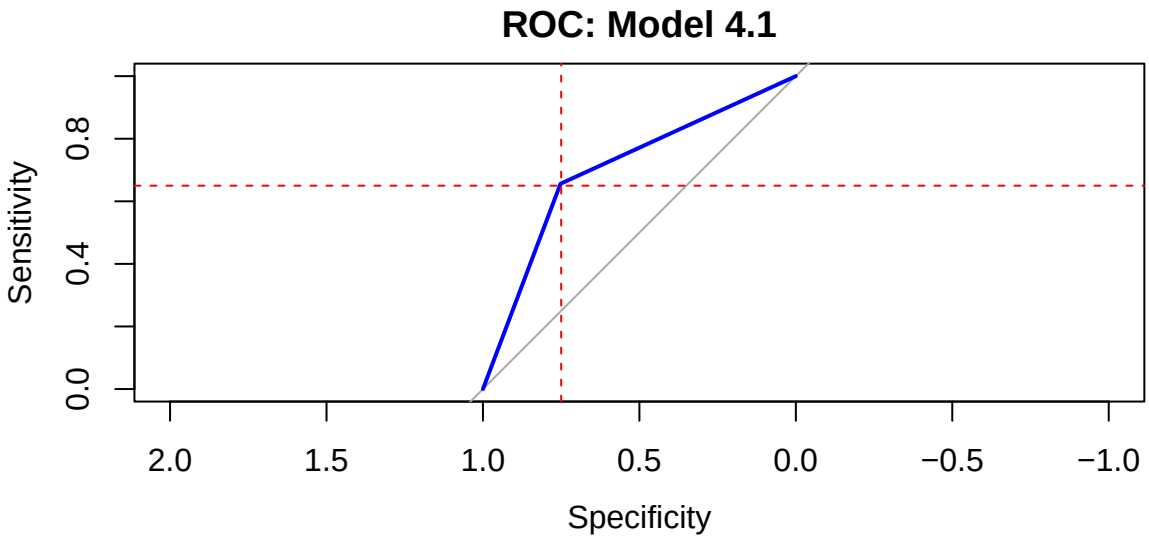


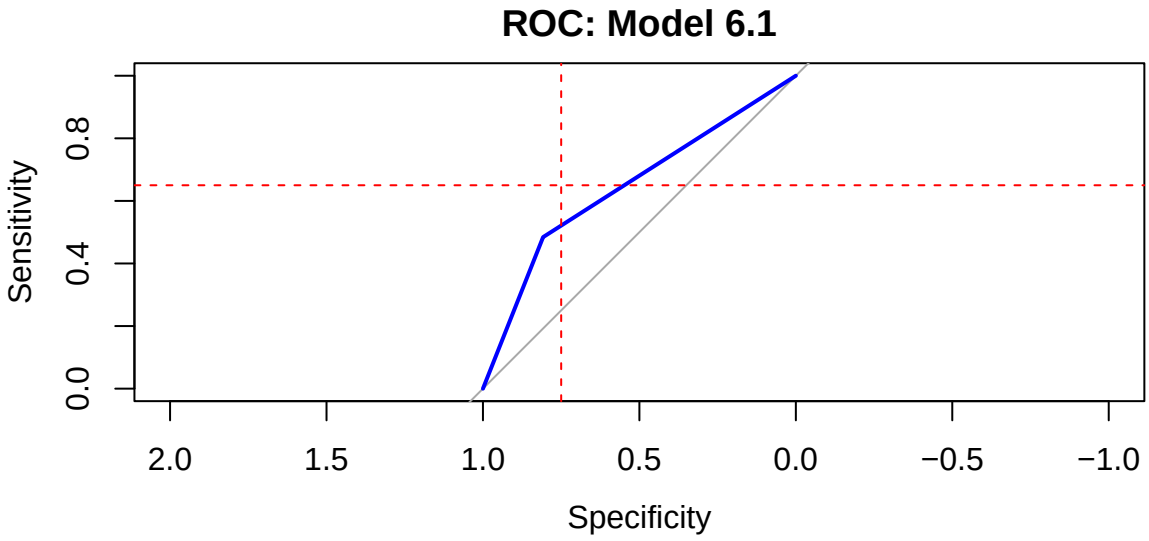

ROC: Model 1.2
Sensitivity
0.0
0.4
0.8
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Specificity


ROC: Model 2.1
Sensitivity
0.0
0.4
0.8
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Specificity


ROC: Model 2.2
Sensitivity
0.0
0.4
0.8
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Specificity


ROC: Model 3.1
Sensitivity
0.0
0.4
0.8
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Specificity


ROC: Model 4.1
Sensitivity
0.0
0.4
0.8
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Specificity


ROC: Model 6.1
Sensitivity
0.0
0.4
0.8
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Specificity